A 4.2BSD Interprocess Communication Primer
DRAFT of January 22, 1983

Samuel J. Leffler

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720
(415) 642-7780

ABSTRACT

This document contains an introduction to the interprocess communication facilities included in the 4.2bsd release of the VAX* UNIX** system, as distributed by U. C. Berkeley.

It discusses the overall model for interprocess communication and introduces the interprocess communication primitives which have been added to the system. The majority of the document considers the use of these primitives in developing applications. The reader is expected to be familiar with the C programming language as all examples given are written in C.

ucbvax!sam

• DEC and VAX are trademarks of Digital Equipment Corporation.
•• UNIX is a Trademark of Bell Laboratories.

1. INTRODUCTION

One of the most important parts of 4.2bsd is the interprocess communication facilities. These facilities are the result of more than two years of discussion and research. The facilities provided in 4.2bsd incorporate many of the ideas from current research, while trying to maintain the UNIX philosophy of simplicity and conciseness. The interprocess communication facilities included in 4.2bsd are intended to establish a standard for UNIX. From the response to the design, [1], it appears many organizations carrying out work with UNIX appear to be adopting the design.

UNIX has previously been very weak in the area of interprocess communication. Prior to the 4.2bsd facilities, the only standard mechanism which allowed two processes to communicate has been pipes (the mpx files which were part of Version 7 were experimental). Unfortunately, pipes are very restrictive in that the two communicating processes must be related through a common ancestor. Further, the semantics of pipes makes them almost impossible to extend to a distributed environment.

Earlier attempts at extending the ipc facilities of UNIX have met with mixed reaction. The majority of the problems have been related to the fact these facilities have been tied to the UNIX file system; either through naming, or implementation. Consequently, the ipc facilities provided in 4.2bsd have been designed as a totally independent subsystem. The 4.2bsd ipc allows processes to rendezvous in many ways. Processes may rendezvous through a UNIX file system-like name space (a space where all names are path names) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Further, the communication facilities have been extended to included more than the simple byte stream provided by a pipe-like entity. These extensions have resulted in a completely new part of the system which users will need time to familiarize themselves with. It is likely that as more use us made of these facilities they will be refined, only time will tell.

The remainder of this document is organized in five sections. Section 2 introduces the new system calls and the basic model of communication. Section 3 describes some of the supporting library routines users may find useful in constructing distributed applications. Section 4 is concerned with the client/server model used in developing applications and includes examples of the two major types of servers. Section 5 considers some of the issues related to protection. Section 6 delves into advanced topics which sophisticated users are likely to encounter when using the ipc facilities.

Why?
- distributed applications
- network
- resource sharing
- distributed file system

Naive - No seeks, ioctl
- stdin, stdout

Sophisticated - shells, screen edit
window managers

30 VAXes at Berkeley

"We don't know what we're doing"

∴ extensible

2. BASICS

The basic building block of communication is the *socket*. A socket is an endpoint of communication to which a name may or may not be *bound*. Each socket in use has a *type* and one or more associated processes. Sockets exist within *communication domains*. A communication domain is an abstraction introduced to bundle the underlying assumptions upon which communication between sockets is predicated. One property emobodied in a communication domain is the scheme used in naming sockets. For example, in the UNIX communication domain sockets are currently named with UNIX path names; e.g. a socket may be named "/dev/foo". Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation processs performed). The 4.2bsd ipc supports two separate communication domains: the UNIX domain, and the Internet domain for processes which communicate using the the DARPA standard communication protocols. The underlying communication facilities provided by these domains have a significant influence on the internal system implementation, as well as the interface to socket facilities provided a user. A notable example of the latter is that a socket "operating" in the UNIX domain sees a subset of the possible error conditions which are possible when operating in the Internet domain.

Typing of sockets is also an important notion. Sockets have been typed according to the communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, though in a distributed environment there is nothing that prevents communication between sockets of different types, provided, of course, the underlying communications protocols support this.

Three types of sockets currently are available to a user. A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of stream sockets provides an interface nearly identical to that of pipes*.

Must make connection - out of band sign

A *datagram* socket supports bidirectional flow of data which is not promised to be sequenced or unduplicated. That is, a process receiving messages on a datagram socket may find data replicated and, possibly, arrive out of sequence. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in most contemporary packet switched networks such as the Ethernet.

A *raw* socket allows users to access the underlying communication protocols used to support the basic socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. The use of raw sockets will be considerd in section 6.

Two potential socket types which have interesting properties are the *sequenced packet* socket and the *reliably delivered message* socket. A sequenced packet socket is identical to a stream socket with the exception that record boundaries are preserved. This interface is very similar to that provided by the Xerox NS Sequenced Packet protocol. The reliably delivered message socket has the same properties as a datagram socket, but with reliable delivery. While we have loosely defined these two socket types, they are currently unimplemented in 4.2bsd. As such, we will concern ourselves only with the three socket types for which support exists.

To create a socket the *socket* system call is used:

```
s = socket(domain, type, protocol);
```

This call requests the system create a socket in the specified *domain* and of the specified *type*. A

---

* In the UNIX domain, in fact, the semantics are identical and, as one might expect, pipes have been implemented internally as simply a pair of connected sockets

particular protocol may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type. The user is returned a descriptor (a small integer number) which may be used in later ipc related system calls. The domain is specified as one of the manifest constants defined in the file <sys/socket.h>. For the UNIX domain the constant is AF_UNIX; for the Internet domain AF_INET. The socket types are also defined in this file and one of SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW must be specified. To create a stream socket in the Internet domain the following call might be used:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for on-machine use a sample call might be:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

To obtain a particular protocol one selects the protocol number, as defined within the communication domain. For the Internet domain the available protocols are defined in <netinet/in.h> or, better yet, one may use one of the library routines which are discussed in section 3. The routine *getprotobyname* may be used in conjunction with the socket call as follows:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
...
pp = getprotobyname("tcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

There are several reasons a socket call may fail. Aside from the rare occurence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

Once a socket has been created, one may perform many operations on it. The most common of these is to bind a name to the socket. Socket which are to be used as "queueing points" in a rendezvous with a client process must bind a name to their socket; otherwise the system would have no way of identifying the server. The *bind* call is used for this:

```
bind(s, name, namelen);
```

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). Its interpretation varies from communication domain to communication domain (this is one of the properties which comprise the "domain"). In the UNIX domain names are path names while in the Internet domain names contain an Internet address and port number. If one wanted to bind a UNIX domain socket to the name "/dev/foo", the following would be used:

```
bind(s, "/dev/foo", sizeof ("/dev/foo"));
```

(Note how the null byte in the name is not counted as part of the name.) In binding an Internet address things become more complicated. The actual call is simple,

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
bind(s, &sin, sizeof (sin));
```

but the selection of what to place in the address *sin* requires some discussion. We will come back

to the problem of formulating Internet addresses in section 3 when the library routines used in name resolution are discussed.

With a bound socket it is possible to rendezvous with an unrelated process. This operation is usually asymetric with one process a "client" and the other a "server". The client requests services from the server by initiating a "connection" to the server's socket. The server, when willing to offer its advertised services, passively "listens" on its socket. On the client side the *connect* call is used to initiate a connection. Using the UNIX domain, this might appear as,

```
connect(s, "server-name", sizeof ("server-name"));
```

while in the Internet domain,

```
connect(s, &server, sizeof (server));
```

If the socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket. An error is returned when the connection was unsuccesful. Otherwise, the socket is associated with the server and data transfer may begin.

Many errors may be returned when a connection attempt fails, we consider the most common:

EISCONN
: The socket is already connected. The current connection is not affected.

ETIMEDOUT
: The protocol decided there was no point in retrying the connection attempt anymore. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost.

ECONNREFUSED
: The host refused service for some reason. When connecting to a host running 4.2bsd this due to the server process not being present at the requested name.

ENETDOWN or EHOSTDOWN
: These operational errors are returned based on status information delivered to the client host by the underlying communication services.

ENETUNREACH or EHOSTUNREACH
: These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to distinguish a network being down from a host being down. In these cases the system is conservative and indicates the entire network is unreachable.

For the server to receive a client's connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the *listen* call specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server process. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual packets which comprise the request will be dropped. This gives a harried server time to make room in its pending connection queue while the client retrys the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though it is unlikely. The backlog figure supplied with the listen call is quietly chopped by the system to allow a maximum of 5 pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then refusing to ignoring all requests.

With a socket marked as listening, a server may *accept* a connection:

```
fromlen = sizeof (from);
snew = accept(s, &from, &fromlen);
```

A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name.

Accept is normally a blocking call. That is, the call to accept will not return until a connection is available or the system call is interrupted by a signal to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or not block on the accept call there are alternatives; they will be considered in section 6.

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect in this case, the normal *read* and *write* system calls are useable,

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to *read* and *write*, the new calls *send* and *recv* are may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While *send* and *recv* are vitually identical to *read* and *write*, the extra *flags* argument is important. The flags may be specified as a non-zero value if one or more of the following is required:

SOF_OOB	send/receive out of band data
SOF_PREVIEW	look at data without reading
SOF_DONTROUTE	send data without routing packets

Out of band data is a notion specific to stream sockets, and one which we will not immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When SOF_PREVIEW is specified with a *recv* call, any data present is returned to the user, but treated as still "unread". That is, the next *read* or *recv* call applied to the socket will return the data previously previewed.

Once a socket is no longer of interest, it may be discarded by applying a *close* to the descriptor,

```
close(s);
```

If data is associated with a socket which promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user find no use for any pending data, it may perform a *shutdown* on the socket prior to closing it. This call is of the form:

```
shutdown(s, &how);
```

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received. Applying shutdown to a socket causes any data queued to be immediately discarded.

To this point we have been concerned mostly with sockets which follow a connection oriented model. However, there is also support for connectionless interactions typical of the

datagram facilities found in contemporary packet switched networks. A datagram socket provides a symetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before, and each should have a name bound to it in order that the receipient of a message may identify the sender. However, to send data the *sendto* primitive must be used,

```
sendto(s, buf, buflen, flags, &to, tolen);
```

The *flags* parameter is used as before. The *to* and *tolen* values are used to indicate the intended receipient of the message. When using an unreliable datagram interface, it is unlikely any errors will be reported to the sender. Where information is present locally to recognize a message which may never be delivered (for instance when a network is unreachable), the caller will be informed.

To receive messages on an unconnected datagram socket, the *recvfrom* primitive is provided:

```
recvfrom(s, buf, buflen, flags, &from, &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer.

In addition to the two calls mentioned above, datagram sockets may also use the *connect* call to associate a socket with a specific address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Other of the less important details of datagram sockets will be described in section 6.

One last basic facility required to develope applications is the ability to multiplex i/o requests among multiple sockets and/or files. This is done using the *select* call:

```
select(nfds, &readfds, &writefds, &execptfds, &timeout);
```

Select takes as arguments three bit masks, one for the set of file descriptors for which the caller wishes to be able to read data on, one for those descriptors to which data is to be written, and one for which exceptional conditions are pending. Bit masks are created by or-ing bits of the form "1 << fd". That is, a descriptor *fd* is selected if a 1 is present in the *fd*'th bit of the mask. The parameter *nfds* specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) specified in a mask. A timeout value may be specified if the selection is not to last more than a predetermined period of time. If *timeout* is set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer (0), the selection will block indefinitely*. *Select* provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in section 6.

also non-blocking I/O on read/write/accept + connect

done with ioctl(, FIOBIO, &1/0)

* To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

3. LIBRARY ROUTINES

The discussion in section 2 indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. To aid in this task a number of routines have been added to the standard C run-time library. In this section we will consider the new routines provided to deal with network addresses. While the 4.2bsd networking facilities support only the DARPA standard Internet protocols, these routines have been designed with flexibility in mind. As more communication protocols become available, we hope the same user interface will be maintained in accessing network related address data bases. The only difference should be the values returned to the user. Since these values are normally just supplied the system, users should not need to be directly aware of the communication protocol and/or naming conventions in use.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name which is intended for human consumption; e.g. "the *login server* on host monet". This name, and the name of the peer host, must then be translated into network *addresses* which are not necessarily suitable for human consumption. Finally, the address must then used in locating a physical *location* and *route* to the service. The specifics of these three mappings is likely to vary between network architectures. For instance, it is desirable for a network to not require hosts be be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows hosts to be physically mobile without requiring it notify its clientele of its current location.

Standard routines are provided for: mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file <netdb.h> must be included when using any of these routines.

A host name to address mapping is represented by the *hostent* structure:

```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    *h_addr;        /* address */
};
```

The official name of the host and its public aliases are returned, along with a variable length address and address type. The routine *gethostbyname*(3N) takes a host name and returns a hostent structure, while the routine *gethostbyaddr*(3N) maps host addresses into a hostent structure. It is possible for a host to have many addresses, all having the same name. Gethostybyname returns the first matching entry in the data base file */etc/hosts*; if this is unsuitable, the lower level routine *gethostent*(3N) may be used. For example, to obtain a *hostent* structure for a host on a particular network the following routine might be used (only Internet addresses are considered, for simplicity):

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
...
struct hostent *
gethostbynameandnet(name, net)
        char *name;
        int net;
{
        register struct hostent *hp;
        register char **cp;

        while ((hp = gethostent()) != NULL) {
                if (hp->h_addrtype != AF_INET)
                        continue;
                if (strcmp(name, hp->h_name)) {
                        for (cp = hp->h_aliases; cp && *cp != NULL; cp++)
                                if (strcmp(name, *cp) == 0)
                                goto found;
                        continue;
                }
                if (in_netof(*(struct in_addr *)hp->h_addr)) == net)
                        break;
        }
        return (hp);
}
```

(*in_netof*(3N) is a standard routine which returns the network portion of an Internet address.)

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
 */
struct  netent {
        char        *n_name;        /* official name of net */
        char        **n_aliases;    /* alias list */
        int         n_addrtype;     /* net address type */
        int         n_net;                          /* network # */
};
```

The routines *getnetbyname*(3N), *getnetbynumber*(3N), and *getnetent*(3N) are the network counterparts to the host routines described above.

For protocols the *protoent* structure defines the protocol-name mapping used with the routines *getprotobyname*(3N), *getprotobynumber*(3N), and *getprotoent*(3N):

```
struct  protoent {
        char        *p_name;        /* official protocol name */
        char        **p_aliases;    /* alias list */
        int         p_proto;        /* protocol # */
};
```

Information regarding services is a bit more complicated. A service is expected to reside at a specific "port address" and employ a particular communication protocol. This view is

consistent with the Internet world, but inconsistent with other network architectures. Further, a service may reside on multiple ports or support multiple protocols. If either of these occurs, the higher level library routines will have to be bypassed in favor of homegrown routines similar in spirit to the "gethostbynameandnet" routine described above. A service mapping is described by the *servent* structure,

```
struct  servent {
        char    *s_name;        /* official service name */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port # */
        char    *s_proto;       /* protocol to use */
};
```

The routine *getservbyname*(3N) maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

```
sp = getservbyname("telnet", (char *)0);
```

returns the service specification for a telnet server using any protocol, while the call

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server which uses the TCP protocol. The routines *getservbyport*(3N) and *getservent*(3N) are also provided. The *getservbyport* routine has an interface similar to that provided by *getservbyname*; an optional protocol name may be included to qualify lookups.

With the support routines described above, an application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown in Figure 1. (This example will be considered in more detail in section 4.)

If one wanted to make the remote login program independent of the Internet protocols and addressing scheme we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear to be worthwhile. Perhaps when the system is adapted to different network architectures the utilities will be reorganized more cleanly.

Aside from the address-related data base routines, there are several other routines available in the run-time library which are of interest to users. These are intended mostly to simplify manipulation of names and addresses. Table 1 summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values.

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On a VAX, or machine with similar architecture, this is usually reversed. Consequently, programs are sometimes required to byte swap quantities. The library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed out the following code would be required:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
...
main(argc, argv)
        char *argv[];
{
        struct sockaddr_in sin;
        struct servent *sp;
        struct hostent *hp;
        int s;
        ...
        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogin: tcp/login: unknown service\n");
                exit(1);
        }
        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }
        bzero((char *)&sin, sizeof (sin));
        bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
        sin.sin_family = hp->h_addrtype;
        sin.sin_port = sp->s_port;
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }
        ...
        if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
                perror("rlogin: connect");
                exit(5);
        }
        ...
}
```

Figure 1. Remote login client code.

Call	Synopsis
bcmp(s1, s2, n)	compare byte-strings; 0 if same, not 0 otherwise
bcopy(s1, s2, n)	copy n bytes from s1 to s2
bzero(base, n)	zero-fill n bytes starting at base
htonl(val)	convert 32-bit quantity from VAX to network byte order
htons(val)	convert 16-bit quantity from VAX to network byte order
ntohl(val)	convert 32-bit quantity from network to VAX byte order
ntohs(val)	convert 16-bit quantity from network to VAX byte order

Table 1. C run-time routines.

```
printf("port number %d\n", ntohs(sp->s_port));
```

On machines other than the VAX these routines are defined as null macros. In hindsight it seems clear the system should be responsible for any byte swapping problems, with the user dealing only with network addresses and numbers in the machine representation. If this is eventually changed, programs which use the library routines described here should only require recompilation, unless they perform explicit manipulation of addresses (as above).

4. CLIENT/SERVER MODEL

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymetry in establishing communication between the client and server which has been examined in section 2. In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

Client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may be symetric or asymetric. In a symetric protocol, either side may play the client or server roles. In an asymetric protocol, one side is immutably recognized as the server, with the other the client. An example of a symetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symetric or asymetric, when accessing a service there is a "client process" and a "server process". We will first consider the properties of server processes, then client processes.

A server process normally listens at a well know address for service requests. Alternative schemes which use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. The Xerox Courier protocol uses the latter scheme. When using Courier, a Courier client process contacts a Courier server at the remote host and identifies the service it requires. The Courier server process then creates the appropriate server process based on a data base and "splices" the client and server together, voiding its part in the transaction. This scheme is attractive in that the Courier server process may provide a single contact point for all services, as well as carrying out the initial steps in authentication (for more about authentication see section 5). However, while this is an attractive possibility for standardizing access to services, it does introduce a certain amount of overhead due to the intermediate process involved. Implementations which provide this type of service within the system can minimize the cost of client server rendezvous . The *portal* notion described in the "4.2BSD System Manual" embodies many of the ideas found in Courier, with the rendezvous mechanism implemented internal to the system.

In 4.2bsd most servers are accessed at well known Internet addresses or UNIX domain names. When a server is started at boot time it advertises it services by listening at a well know location. For example, the remote login server's main loop is of the form shown in Figure 2.

The first step taken by the server is look up its service definition:

```
main(argc, argv)
        int argc;
        char **argv;
{
        int f;
        struct sockaddr_in from;
        struct servent *sp;

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogind: tcp/login: unknown service0);
                exit(1);
        }
        ...
#ifndef DEBUG
        <<disassociate server from controlling terminal>>
#endif
        ...
        sin.sin_port = sp->s_port;
        ...
        f = socket(AF_INET, SOCK_STREAM, 0, 0);
        ...
        if (bind(f, (caddr_t)&sin, sizeof (sin)) < 0) {
                ...
        }
        ...
        listen(f, 10);
        for (;;) {
                int g, len = sizeof (from);

                g = accept(f, &from, &len, 0);
                if (g < 0) {
                        if (errno != EINTR)
                                perror("rlogind: accept");
                        continue;
                }
                if (fork() == 0) {
                        close(f);
                        doit(g, &from);
                }
                close(g);
        }
}
```

Figure 2. Remote login server.

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service0);
        exit(1);
}
```

This definition is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

Step two is to disassociate the server from the controlling terminal of its invoker. This important as the server will likely not want to receive signals delivered to the process group of the

controlling terminal.

```
#ifndef DEBUG
        if (fork())
                exit(0);
        for (f = 0; f < 10; f++)
                (void) close(f);
        (void) open("/", 0);
        (void) dup2(0, 1);
        (void) dup2(0, 2);
        { int t = open("/dev/tty", 2);
          if (t >= 0) {
                ioctl(t, TIOCNOTTY, (char *)0);
                (void) close(t);
          }
        }
#endif
```

The steps in this little horror are:

1) Fork once to allow the calling process to continue its business.

2) Close file descriptors to insure the next step opens file descriptors of known value (a descriptor is assigned by the system according to the first available; thus, if descriptor 0 is available, the next open call will always return descriptor 0)

3) Open the root file system directory for reading, then duplicate the descriptor to the standard output and standard error. This is safe (even if the process executes as the super user) because the system disallows any attempts to write on directories). Code of this sort has been carried over from time in memorium and, while most obscure, is known to work (sigh).

4) Clear any process group associated with the server (TIOCNOTTY). If a server allows its process group to be "randomly" assigned, it may receive unexpected signals which could terminate it. By reseting its controlling terminal, it may manipulate its process group in a well defined manner, thereby controlling its own destiny, so to speak.

The steps described are not carried out when the server is compiled for debugging. This allows the user debugging the server to abort servers with an interrupt from the keyboard, and also view diagnostic output normally lost because the standard output and error descriptors have been closed. A better scheme whereby diagnostics are sent, in messages, to a system logger process is planned for the future.

Once a server has established a pristine evironment, it creates a socket and begins accepting service requests. The *bind* call is required to insure the server listens at its expected location. The main body of the loop is fairly simple:

```
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, &from, &len, 0);
        if (g < 0) {
                if (errno != EINTR)
                        perror("rlogind: accept");
                continue;
        }
        if (fork() == 0) {
                close(f);
                doit(g, &from);
        }
        close(g);
}
```

because of SIGCHLD

An *accept* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as SIGCHLD (to be discussed in section 6). Therefore, the return value from *accept* is checked to insure a connection has really been established. With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queueing connection requests is closed in the child, while the socket created as a result of the accept is closed in the parent. The address of the client it also handed the *doit* routine because it requires it in authenticating clients.

The client side of the remote login service was shown earlier in Figure 1. One can see the separate, asymetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let's consider more closely the steps taken by the client remote login process. As in the server process, the first step is to locate the service definition for a remote login.

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogin: tcp/login: unknown service\n");
        exit(1);
}
```

Next the destination host is looked up with a *gethostbyname* call:

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
        fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is zeroed, then filled in with the Internet address of the foreign host and the port number at which the login process resides:

```
bzero((char *)&sin, sizeof (sin));
bcopy(hp->h_addr, (char *)sin.sin_addr, hp->h_length);
sin.sin_family = hp->h_addrtype;
sin.sin_port = sp->s_port;
```

A socket is created, and a connection initiated.

```
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }
        ...
        if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
                perror("rlogin: connect");
                exit(4);
        }
```

many ucb machines use only rlogin, have no physical ttys
CPU cost comparable
1 server process/line
3M Ether
4 4.1c
rest 4.1a

The details of the remote login protocol will not be considered here; that part of the initial connection protocol which relates to authenticating the client will be considered in section 5.

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the "rwho" service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to *broadcast* information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

An user on any machine running the rwho server may find out the current status of a machine with the *ruptime*(1) program. The output generated is illustrated in Figure 3.

```
arpa    up 9:45,                5 users, load 1.15, 1.39, 1.31
cad     up 2+ 12:04,    8 users, load 4.67, 5.13, 4.59
calder  up 10:10,       0 users, load 0.27, 0.15, 0.14
dali    up 2+ 06:28,    9 users, load 1.04, 1.20, 1.65
degas   up 25+ 09:48,   0 users, load 1.49, 1.43, 1.41
ear     up 5+ 00:05,    0 users, load 1.51, 1.54, 1.56
ernie   down 0:24
esvax   down 17:04
ingres  down  0:26
kim     up 3+ 09:16,    8 users, load 2.03, 2.46, 3.11
matisse up 3+ 06:18,    0 users, load 0.03, 0.03, 0.05
medea   up 3+ 09:39,    2 users, load 0.35, 0.37, 0.50
merlin  down 19+ 15:37
miro    up 1+ 07:20,    7 users, load 4.59, 3.28, 2.12
monet   up 1+ 00:43,    2 users, load 0.22, 0.09, 0.07
oz      down 16:09
statvax up 2+ 15:57,    3 users, load 1.52, 1.81, 1.86
ucbvax  up 9:34,                2 users, load 6.08, 5.16, 3.28
```

Figure 3. ruptime output.

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

The rwho server, in a simplified form, is pictured in Figure 4. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to insure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an exended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error, but serves our current needs (one should not, however, depend on this information for a true host status).

```
main()
{
        ...
        sp = getservbyname("who", "udp");
        net = getnetbyname("localnet");
        sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
        sin.sin_port = sp->s_port;
        ...
        s = socket(AF_INET, SOCK_DGRAM, 0);
        ...
        bind(s, &sin, sizeof (sin));
        ...
        sigset(SIGALRM, onalrm);
        onalrm();
        for (;;) {
                struct whod wd;
                int cc, whod, len = sizeof (from);

                cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0, &from, &len);
                if (cc <= 0) {
                        if (cc < 0 && errno != EINTR)
                                perror("rwhod: recv");
                        continue;
                }
                if (from.sin_port != sp->s_port) {
                        fprintf(stderr, "rwhod: %d: bad from port\n",
                                ntohs(from.sin_port));
                        continue;
                }
                ...
                if (!verify(wd.wd_hostname)) {
                        fprintf(stderr, "rwhod: malformed host name from %x\n",
                                from.sin_addr);
                        continue;
                }
                (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
                whod = creat(path, 0666);
                ...
                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

Figure 4. rwho server.

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a packet and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet does, however, indicates some problems with the current protocol.

Status information is **broadcast** on the **local network**. For networks which do not support the notion of broadcast another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status received). This,

unfortunately, requires some "bootstrapping" information, as a server started up on a "quiet" network will have no known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied it, status information may then propagate through a neighbor to hosts which are not (possibly) directly neighbors. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitary topology may share status information*.

The second problem with the current scheme is that the rwho process services only a single local network, and this network is found by reading a file. It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. 4.2bsd attempts to isolate host-specific information from applications by providing system calls which return the necessary information†. Unfortunately, no straightforward mechanism currently exists for finding the collection of networks to which a host is directly connected. Thus the rwho server performs a lookup in a file to find its local network. A better, though still unsatisfactory, scheme used by the routing process is to interrogate the system data structures to locate those directly connected networks. A mechanism to acquire this information from the system is likely to be added to the system.

---

* One must, however, be concerned about "loops". That is, if a host is connected to multiple networks, it will receive status information from itself. It must be careful not to consider itself as a neighbor and attempt to actively exchange information with itself for this would result in an endless exchange of messages.
† An example of such a system call is the *gethostnamd*(2) call which returns the host's "official" name.

5. AUTHENTICATION AND SECURITY

In a distributed environment processes often must know the identity of a client. This information may be used for access control, for accounting, etc. The problem of authenticating client processes in such an environment has been researched extensively with two major schemes developed. The first is based on the involvement of a trusted partner. One process is informed of the identity of its client by some trusted party such as the operating system. In shifting the problem from client/server to a single, but possibly distributed, entity one arrives at a model based on capability, or rights, being passed *internal* to a system.

The alternative to internalized capabilities is to require each process maintain its own rights external to any partner such as the operating system. Authentication is then performed by a server verifying the acceptability of capabilities held by a client. Schemes of this sort require that the information used for authentication purposes be encrypted to prevent unwarranted theft of service capabilities.

Schemes based on internalized capabilities inevitably rely on externalized capabilities at some level for authenticating each distributed part of the trusted entity. No matter whether internalized or externalized capabilities are used for authenticating clients, once a connection has been established messages exhanged between trusted parties must be safeguarded against unwarranted *eavesdropping* and *spoofing*. All this requires that for a secure environment encryption techniques must be used. Unfortunately, the available hardware for performing encryption is far too slow to provide acceptable performance in constructing a secure distributed environment.

Given the lack of hardware support for providing secure externalized (or internalized) capabilities in a distributed environment, 4.2bsd makes no attempt at providing such a facility. Instead, the services developed allow users to construct applications which use a restricted form of internalized capabilities based on the belief that a network is *physically secure*. That is, most of the services which are part of 4.2bsd assume that clients access a network through another, equally trusted, system. This assumption is obviously faulty on a network including personal workstations, for network integrity is easily compromised by a user at a workstation subverting the local authentication mechanisms in order to masquerade as another identity.

This section will consider those authentication techniques used by the 4.2bsd standard services, and consider their limitations. One of the goals of 4.2bsd is to provide a basis for continued research in security in a distributed environment. As practical solutions are developed they will be incorporated in the system*.

There are two schemes used for authenticating clients. The first requires the client to essentially execute a login before gaining access to a service. That is, the client must provide a user name and password (encypted) on the remote host. This information is then verified against the information stored in the public *passwd*(5) file. If the validation succeeds, the requested services are made available and remote execution takes place in the protection domain of the user's account (i.e. the server performs *setuid* and *setgid* calls to establish the server's protection to be that of the supplied user). This form of authentication is as safe as the password supplied, but is slow due to the encryption required to authenticate passwords. The *rexec*(3) library routine provides a standard facility for executing commands on remote machines and using passwords as an authentication mechanism.

The second form of authentication is based on the assumption the network is physically secure. In this scheme all port numbers in the range 0-1023 are reserved for the super user. That is, only a super user may bind a port number to a socket in this range. By depending on this fact, and the assumption the cable is free of spoofing, the server side then verifies it is negotiating with a trusted client process by checking the port number of the source address of a connection

---

* For now, users concerned about the safety of information accessable in a distributed environment should keep their files encrypted, and/or not run those servers which authentication mechanism described in this section.

(or received packet). If the source address indicates it includes a port number in the priviledged range, any information passed to the server is believed to be correct. The remote login and remote shell servers utilize code of the form shown in Figure 5 for this purpose. The client's address is first checked to verify it is associated with a known host (i.e. that it is present in the host name data base). Then, the source port and address format are checked for correctness. Should any of the checks fail, the server refuses service.

```
doit(f, fromp)
        int f;
        struct sockaddr_in *fromp;
{
        register struct hostent *hp;
        ...
        fromp->sin_port = htons((u_short)fromp->sin_port);
        hp = gethostbyaddr(&fromp->sin_addr, sizeof (struct in_addr),
                        fromp->sin_family);
        if (hp == 0)
                        fatal(f, "Host name for your address unknown");
        if (fromp->sin_family != AF_INET ||
            fromp->sin_port >= IPPORT_RESERVED)
                        fatal(f, "Permission denied");
        ...
}
```

Figure 5. Authentication using reserved ports.

Because the basic authentication mechanism is so lax, secondary security facilities were added to restrict the set of hosts and users that are allowed to use the "authentication based on reserved port numbers" scheme. The file */etc/hosts.equiv* on a client host defines the collection of hosts which are assumed to be under a single administrative control. These hosts must share common user names to allow these names to be passed through, and mapped into user identifiers without confusion. With the exception of the super user (and the *.rhosts* file to be described next), when authentication is performed at the server, if the client host is found in the equivalence file, it is allowed to use reserved port numbers as an authentication mechanism. Otherwise, permission is denied (remote shell execution), or a password is required from the client (remote login). If the intended identity is the super user, or the user has a .rhosts file in their home directory, the information present in the .rhosts file is used to specify an access control list. That is, the .rhosts file contains a list of host-user pairs. If the client host-user identification matches one of the specifications present in .rhosts, authentication continues; otherwise permission is denied. This allows user on equivalenced machines to restrict access to their accounts individually.

6. ADVANCED TOPICS

A number of facilities have yet to be discussed. For most users of the ipc those mechanisms already described will suffice in constructing distributed applications. However, others will find need to utilize some of the less common features which we will consider in this section.

The stream socket abstraction includes the notion of "out of band" data. Out of band data is an entirely separate transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data along with the SIGURG signal. In addition to the information passed, a mark is placed in the data stream to indicate the point at which the out of band data was sent. The remote login and remote shell applications use this facility to propagate signals from between client and server processes. When the signal is expected to flush any pending output from the remote process(es), all data up to the mark in the data stream is discarded. The stream abstraction defines that the out of band data facilities must allow at least one, reliably delivered, out of band message be pending at any time. Further, at least one byte of data must be able to be sent with each out of band notification. To send an out of band message the SOF_OOB flag is supplied to a *send* or *sendto* calls, while to receive out of band data SOF_OOB should be indicated when performing a *recvfrom* or *recv* call. To find out if read pointer is currently pointing at the mark in the data stream, the SIOCATMARK ioctl is provided:

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next read will return data after the mark. Otherwise (assuming out of band data has arrive), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Figure 5.

```
oob()
{
        int out = 1+1;
        char waste[BUFSIZ], mark;

        signal(SIGURG, oob);
        ioctl(1, TIOCFLUSH, (char *)&out);
        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                        perror("ioctl");
                        break;
                }
                if (mark)
                        break;
                (void) read(rem, waste, sizeof (waste));
        }
        recv(rem, &mark, 1, SOF_OOB);
        ...
}
```

Figure 5. Flushing input on receipt of out of band data.

Due to the existence of the SIGURG signal each socket has an associated process group (just as is done for terminals). This process group is initialized to the process group of its creator, but may be redefined at a later time with the SIOCSPGRP ioctl:

```
ioctl(s, SIOCSPGRP, &pgrp);
```

A similar ioctl, SIOCGPGRP, is available for determining the current process group of a socket.

Many programs will not function properly without a terminal for standard input and output. Since a socket is not a terminal, it is often necessary to have a process communicating over the network do so through a *pseudo terminal*. A pseudo terminal is actually a pair of devices, master and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo terminal is supplied as input to a process reading from the master side. Data written on the master side is given the slave as input. In this way, the process manipulating the master side of the pseudo terminal has control over the information read and written on the slave side. The remote login server uses pseudo terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudo terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends an interrupt or quit signal to a process exexcuting on a remote machine, the client login program traps the signal, sends an out of band message to the server process who then uses the signal number, sent as the data value in the out of band message, to perform a *kill*(2) on the appropriate process(es).